

MybatisPlus

I快速入门-简单介绍

常见注解

常见配置

核心功能-条件构造器

[基于QueryWrapper的查询](#)

[基于UpdateWrapper的更新](#)

[什么是lambda语法](#)

自定义sql

Service接口

基于Restful风格实现下列接口

IService的Lambda查询

IService批量新增（批处理）

代码生成（一次生成某表对应的四层框架）

静态工具

逻辑删除

枚举处理器

JSON处理器

插件功能-分页插件基本用法

[分页插件](#)

[通用分页实体](#)

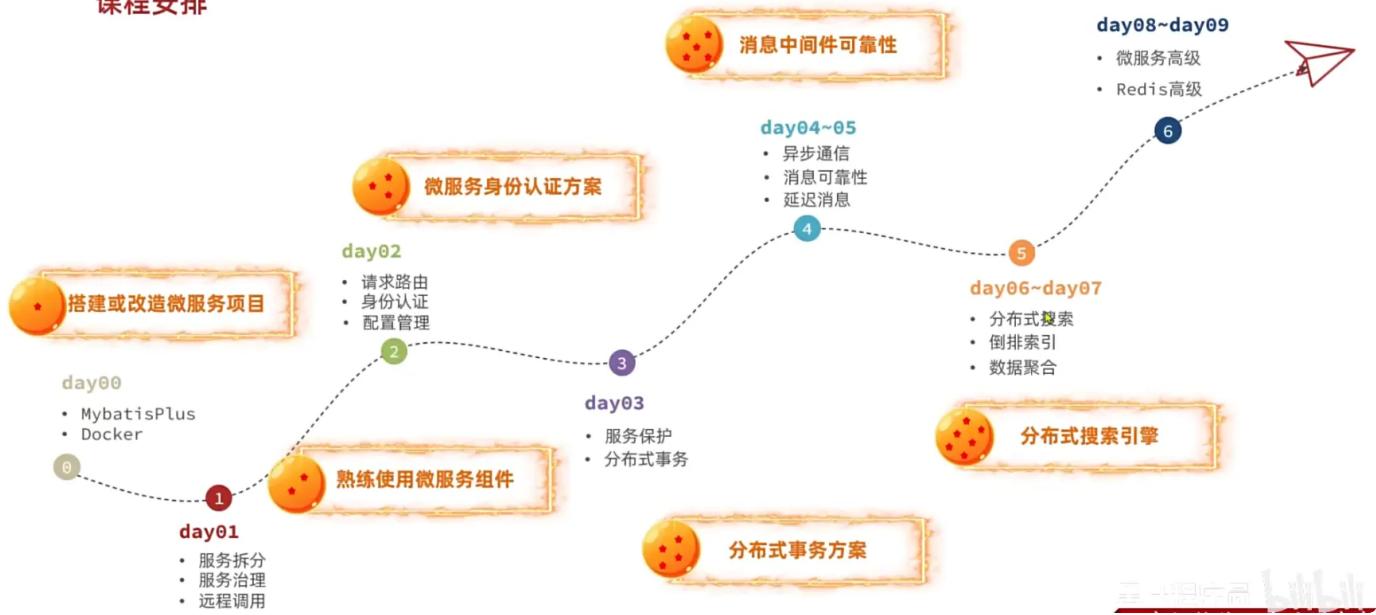
[通用分页实体与MP转换](#)

I快速入门-简单介绍

微服务是一种软件架构风格，它是以专注于单一职责的很多小型项目为基础，组合出复杂的大型应用。
(两只小鸟)



课程安排



1. 引入依赖

这是mybatis的依赖

XML

```

1 <dependency>
2   <groupId>org.mybatis.spring.boot</groupId>
3   <artifactId>mybatis-spring-boot-starter</artifactId>
4   <version>2.3.1</version>
5 </dependency>
    
```

▼ 这是mubatisPlus的依赖

XML |

```
1 <dependency>
2   <groupId>com.baomidou</groupId>
3   <artifactId>mybatis-plus-boot-starter</artifactId>
4   <version>3.5.3.1</version>
5 </dependency>
```

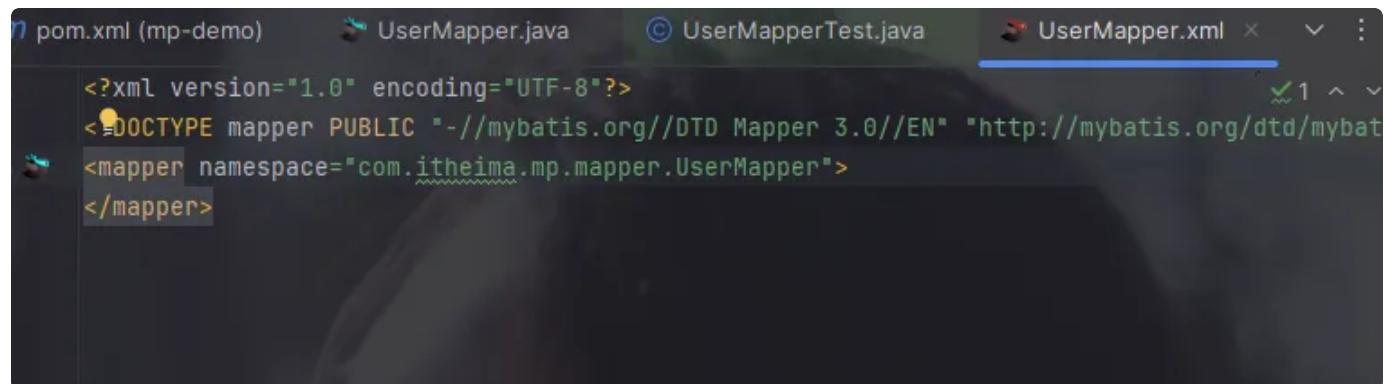
2. 定义Mapper

自定义的Mapper继承MybatisPlus提供的BaseMapper接口：

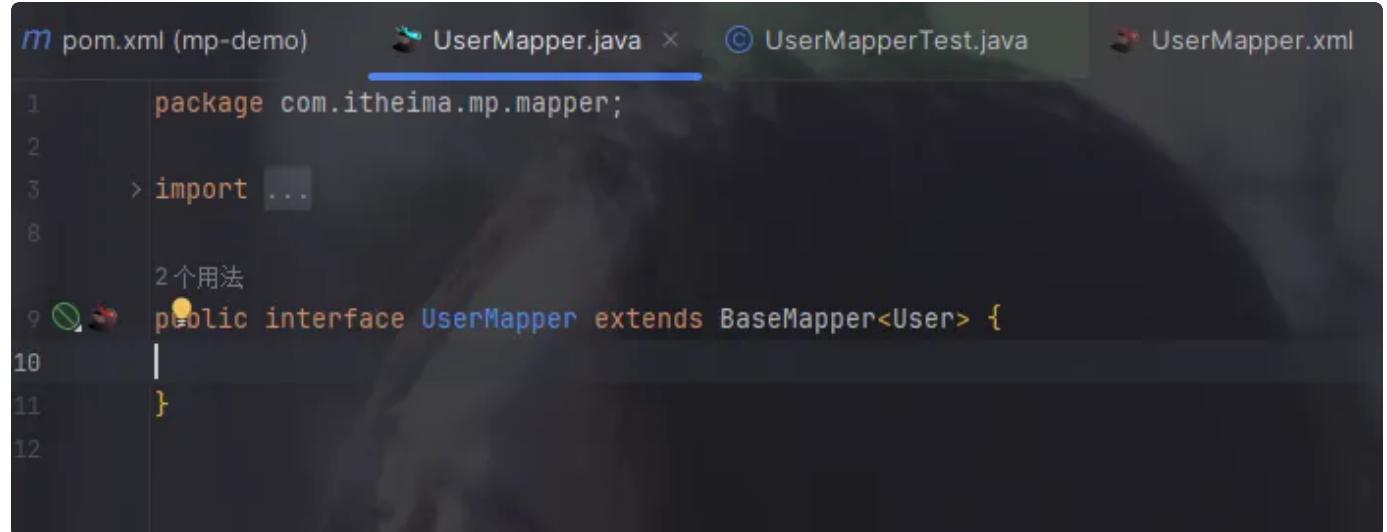
```
public interface UserMapper extends BaseMapper<User> {
}
```

<User>为你实体类的实体，这样才能知道你操作的实体是什么

当添加完mybatisplus后，可以删去红色小鸟的sql语句内容，同样也能删去蓝色小鸟的实现类语句内容



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mp.mapper.UserMapper">
</mapper>
```



```
1 package com.itheima.mp.mapper;
2
3 > import ...
4
5 2个用法
6 <span style="color: #0000ff; font-weight: bold;>@Mapper</span>
7 <span style="color: #0000ff; font-weight: bold;>public interface UserMapper extends BaseMapper<User> {</span>
8
9   <span style="color: #0000ff; font-weight: bold;>    </span>
10  <span style="color: #0000ff; font-weight: bold;>}</span>
11
12
```

直接调用Mapper后会出现很多固定的sql语句提供选择

updateById(User entity)	int
insert(User entity)	int
update(User entity, Wrapper<User> updateWrapper)	int
deleteById(User entity)	int
selectById(Serializable id)	User
equals(Object obj)	boolean
hashCode()	int
toString()	String
delete(Wrapper<User> queryWrapper)	int
deleteBatchIds(Collection<?> idList)	int
deleteById(Serializable id)	int
deleteByMap(Map<String, Object> columnMap)	int
exists(Wrapper<User> queryWrapper)	boolean
selectBatchIds(Collection<? extends Serializable> idList)	List<User>
selectByMap(Map<String, Object> columnMap)	List<User>
selectCount(Wrapper<User> queryWrapper)	Long
selectList(Wrapper<User> queryWrapper)	List<User>
按 Ctrl+. 选择所选 (或第一个) 建议, 然后插入点 下一提示	
	:

以此实现0代码

只需简单的配置, 即可快速进行单表CRUD操作, 从而节省大量的时间

常见注解

MBP通过扫描实体类, 并基于反射获取实体类信息作为数据库表信息

```
public interface UserMapper extends BaseMapper<User> {  
}
```

```
@Data  
public class User {  
    private Long id;  
    private String username;  
    private String password;  
    private String phone;  
    private String info;  
    private Integer status;  
    private Integer balance;  
    private LocalDateTime createTime;  
    private LocalDateTime updateTime;  
}
```

变量名转化形式

- 类名驼峰转下划线作为表名
- 默认把名为id的字段作为主键
- 变量名驼峰转下划线作为表的字段名

MBP提供的注解常用于对应变量值无法探测到，而自行编写的形式

- @ TableName: 用来指定表名
- @TableId :用来指定表中的主键字段信息
- @TableFiled :用来指定表中的普通字段信息

成员变量不是数据库字段

```

@TableName("tb_user")
public class User {
    @TableId(value= "id", type= IdType.AUTO )
    private Long id;
    @TableField("username")
    private String name;
    @TableField("is_married")
    private Boolean isMarried;
    @TableField("`order`")
    private Integer order;
    @TableField(exist = false)
    private String address;
}

```

名称: tb_user	注释: 用户表			
#	名称	数据类型	注释	默认
1	id	BIGINT	用户id	AUTO_INCREMENT
2	username	VARCHAR	用户名	无默认值
3	is_married	BIT	密码	0
4	order	TINYINT	序号	NULL

IdType枚举：

- AUTO : 数据库自增长
- INPUT : 通过set方法自行输入
- ASSIGN_ID : 分配 ID , 接口IdentifierGenerator的方法nextId来生成id ,
默认实现类为DefaultIdentifierGenerator雪花算法

使用@TableField的常见场景：

- 成员变量名与数据库字段名不一致
- 成员变量名以is开头，且是布尔值
- 成员变量名与数据库关键字冲突
- 成员变量不是数据库字段

值曰生虎哥

常见配置

MBP的配置项继承了MB原生配置和一些自己特有的配置。

```

mybatis-plus:
  type-aliases-package: com.itheima.mp.domain.po # 别名扫描包
  mapper-locations: "classpath*/mapper/**/*.xml" # Mapper.xml文件地址, 默认值
  configuration:
    map-underscore-to-camel-case: true # 是否开启下划线和驼峰的映射
    cache-enabled: false # 是否开启二级缓存
  global-config:
    db-config:
      id-type: assign_id # id为雪花算法生成
      update-strategy: not_null # 更新策略: 只更新非空字段

```

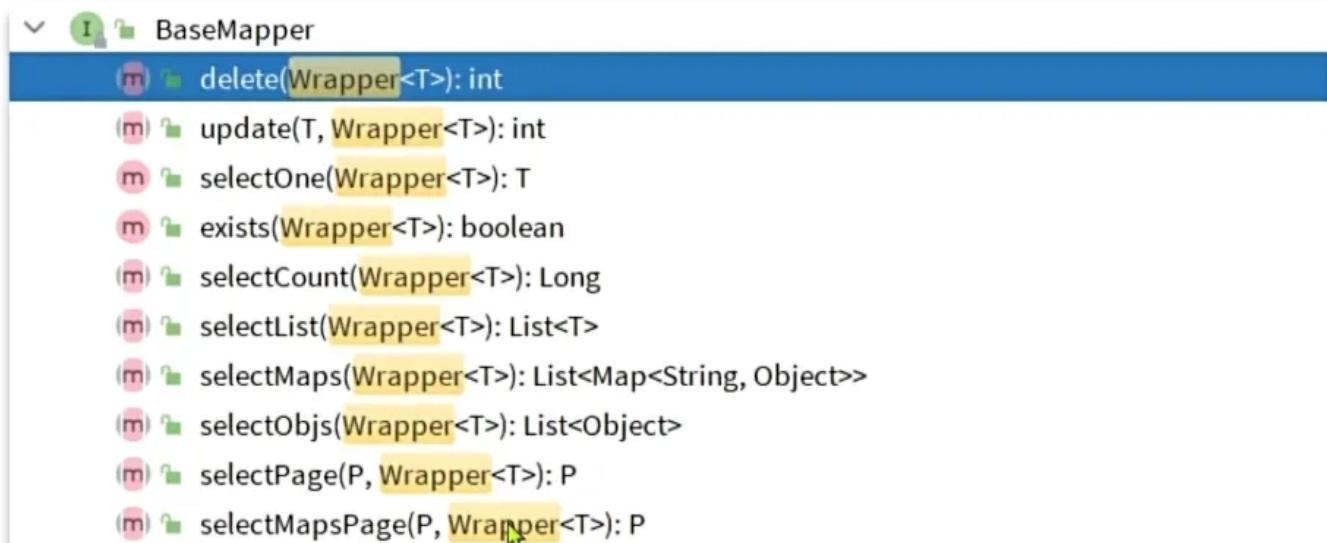
MBP使用基本流程：

- 引入起步依赖
- 自定义Mapper基础BaseMapper
- 在实体类上添加注解声明表信息

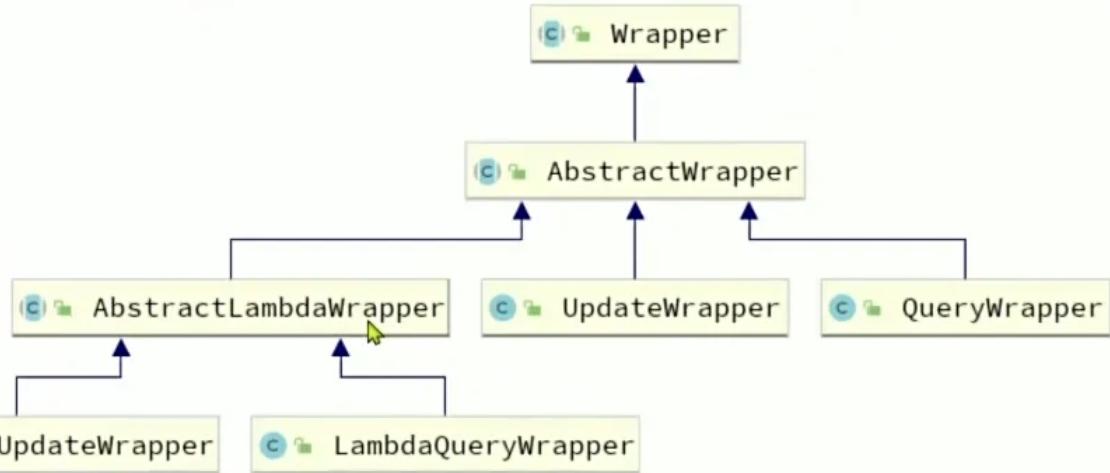
核心功能-条件构造器

MBP支持各种复杂的where条件

MyBatisPlus支持各种复杂的where条件，可以满足日常开发的所有需求。



Wrapper就是条件构造器



基于QueryWrapper的查询

- ① 查询出名字中带o的，存款大于等于1000元的人的id、username、info、balance字段

#	名称	数据类型	注释	长度/集合
1	id	BIGINT	用户id	19
2	username	VARCHAR	用户名	50
3	password	VARCHAR	密码	128
4	phone	VARCHAR	注册手机号	20
5	info	JSON	详细信息	
6	status	INT	使用状态 (1正常 2冻结)	10
7	balance	INT	账户余额	10
8	create_time	DATETIME	创建时间	
9	update_time	DATETIME	更新时间	

● ● ●

SELECT id,username,info,balance
FROM user
WHERE username LIKE ? AND balance >= ?

```

1  @Test
2  void testQueryWrapper() {
3      // 1. 构造查询条件
4      QueryWrapper<User> wrapper = new QueryWrapper<User>()
5          .select("id", "username", "info", "balance")
6          .like("username", "o")
7          .ge("balance", 1000);
8      // 2. 查询
9      List<User> users = userMapper.selectList(wrapper);
10     users.forEach(System.out::println);

```

② 更新用户名为jack的用户的余额为2000



UPDATE user

```

SET balance = 2000
WHERE (username = "jack")

```



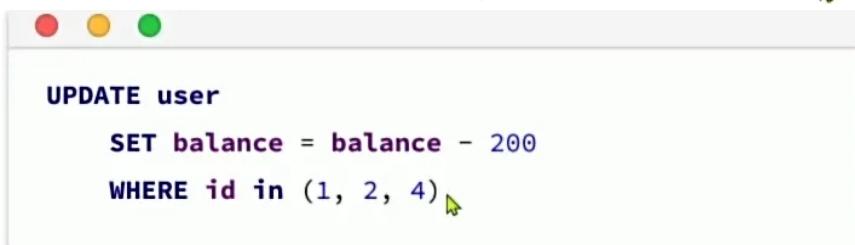
```

1  @Test
2  void testUpdateByQueryWrapper() {
3      // 1. 要更新的数据
4      User user = new User();
5      user.setBalance(2000);
6      // 2. 要更新的条件
7      QueryWrapper<User> wrapper = new QueryWrapper<User>().eq("username", "jack");
8      // 3. 执行更新
9      userMapper.update(user, wrapper);
10 }

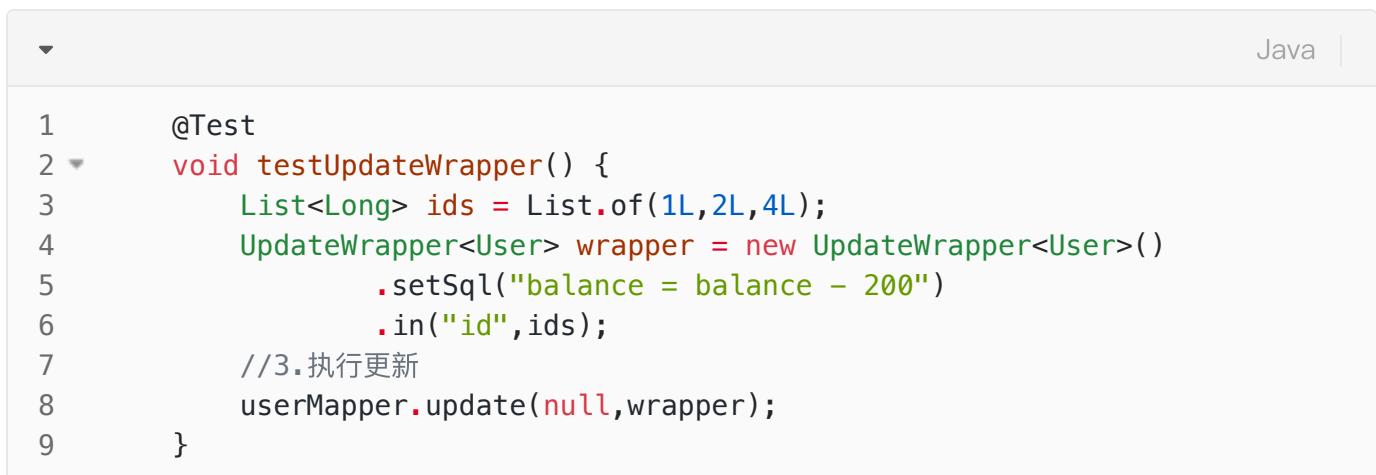
```

基于UpdateWrapper的更新

需求：更新id为1,2,4的用户的余额，扣200



```
UPDATE user
SET balance = balance - 200
WHERE id in (1, 2, 4)
```



```
1 @Test
2 void testUpdateWrapper() {
3     List<Long> ids = List.of(1L, 2L, 4L);
4     UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
5         .setSql("balance = balance - 200")
6         .in("id", ids);
7     //3. 执行更新
8     userMapper.update(null, wrapper);
9 }
```

为了避免硬编码，则使用lambda语法，

什么是lambda语法

```
@Test
void testLambdaQueryWrapper() {
    // 1. 构建查询条件
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>()
        .select(User::getId, User::getUsername, User::getInfo, User::getBalance)
        .like(User::getUsername, val: "o")
        .ge(User::getBalance, val: 1000); | I
    // 2. 查询
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
```

先是在方法名上声明Lambda，或者 .lambda()

接着其中将固定死的属性名转换成 表名：：属性名的形式

条件构造器的用法：

- QueryWrapper和LambdaQueryWrapper通常用来构建select、delete、update的where条件部分
- UpdateWrapper和LambdaUpdateWrapper通常只有在set语句比较特殊才使用
- 尽量使用LambdaQueryWrapper和LambdaUpdateWrapper，避免硬编码

自定义sql

我们可以利用MBP的Wrapper来构造复杂的where条件，然后自己定义sql语句剩下的部分

需求：将id在指定范围的用户（例如1、2、4）的余额扣减指定值



```
<update id="updateBalanceByIds">
    UPDATE user
    SET balance = balance - #{amount}
    WHERE id IN
        <foreach collection="ids" separator="," item="id" open="(" close="")">
            #{id}
        </foreach>
</update>
```

```
@Test  
void testUpdateWrapper() {  
    List<Long> ids = List.of(1L, 2L, 4L);  
    UpdateWrapper<User> wrapper = new UpdateWrapper<User>()  
        .setSql("balance = balance - 200")  
        .in("id", ids);  
    userMapper.update(null, wrapper);  
}
```

1. 基于Wrapper构建where条件

```
List<Long> ids = List.of(1L, 2L, 4L);  
int amount = 200;  
// 1. 构建条件  
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>().in(User::getId, ids);  
// 2. 自定义SQL方法调用  
userMapper.updateBalanceByIds(wrapper, amount);
```

2. 在mapper方法参数中用Param声明wrapper变量名称，必须是ew

```
void updateBalanceByIds(@Param("ew") LambdaQueryWrapper<User> wrapper, @Param("amount") int amount);
```

3. 自定义sql，并使用wrapper条件

```
<update id="updateBalanceByIds">  
    UPDATE tb_user SET balance = balance - #{amount} ${ew.customSqlSegment}  
</update>
```

拼接语句实现sql

Java

```
1  @Test
2  void testCustomSqlUpdate() {
3      //1.更新条件
4      List<Long> ids = List.of(1L,2L,4L);
5      int amount = 200;
6      //2.定义条件
7      QueryWrapper<User> wrapper = new QueryWrapper<User>().in("id",ids)
8      ;
9      //3.调用自定义SQL方法
10     userMapper.updateBalanceByIds(wrapper,amount);
11 }  
//updateBalanceByIds为自定义方法, 需要构造
```

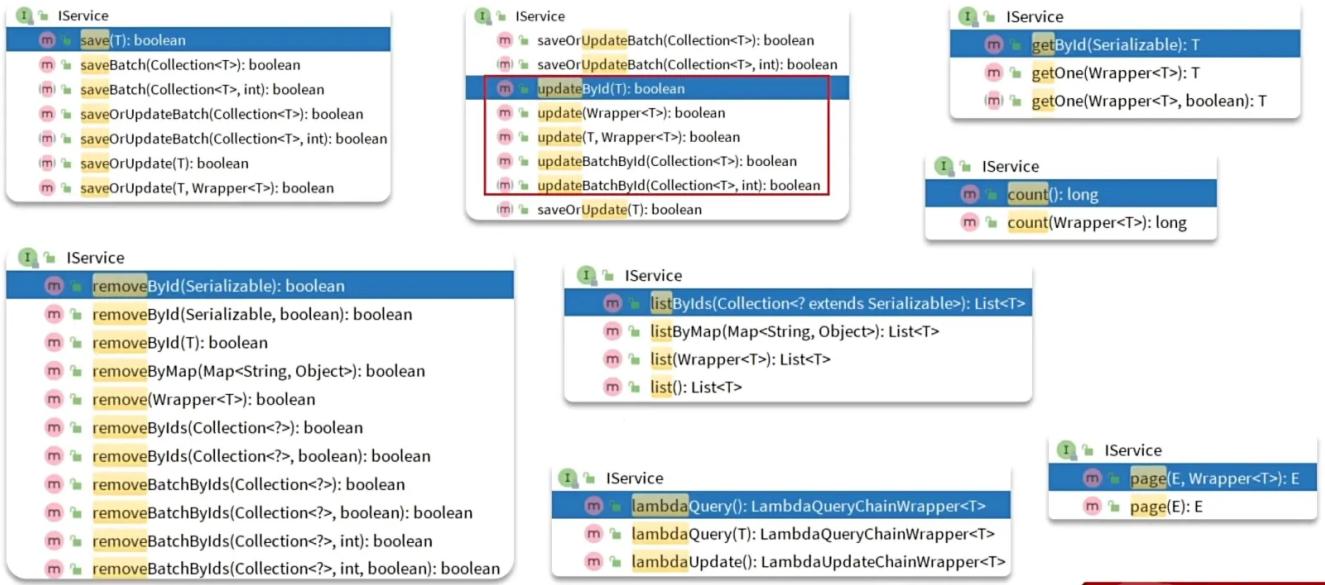
Java

```
1  public interface UserMapper extends BaseMapper<User> {
2      List<User> queryUserByIds(@Param("id") List<Long> ids);
3      //注意加上ew、和注解
4      void updateBalanceByIds(@Param(Constants.WRAPPER) QueryWrapper<User> wrapper,@Param("amount") int amount);
5  }
```

Java

```
1      <update id="updateBalanceByIds">
2          UPDATE user SET balance = balance - #{amount} ${ew.customSqlSegment}
3      </update>
4      //在mapper层编写sql语句
```

Service接口



查询一个就用get，查询多个就用list

新建接口，接口继承IService<实现实体类>

```
2个用法 1个实现
public interface IUserService extends IService<User> {
```

新建IMPL，继承ServiceImpl<实体mapper、实体类>继承接口

```
0个用法
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements IUserService { }
```

基于Restful风格实现下列接口

编号	接口	请求方式	请求路径	请求参数	返回值
1	新增用户	POST	/users	用户表单实体	无
2	删除用户	DELETE	/users/{id}	用户id	无
3	根据id查询用户	GET	/users/{id}	用户id	用户VO
4	根据id批量查询	GET	/users	用户id集合	用户VO集合
5	根据id扣减余额	PUT	/users/{id}/deduction/{money}	<ul style="list-style-type: none"> • 用户id • 扣减金额 	无

引入maven

```

1 <!--swagger-->
2 ▼ <dependency>
3   <groupId>com.github.xiaoymin</groupId>
4   <artifactId>knife4j-openapi2-spring-boot-starter</artifactId>
5   <version>4.1.0</version>
6 </dependency>
7 <!--web-->
8 ▼ <dependency>
9   <groupId>org.springframework.boot</groupId>
10  <artifactId>spring-boot-starter-web</artifactId>
11 </dependency>

```

配置信息

```
1 knife4j:
2   enable: true
3   openapi:
4     title: 用户管理接口文档
5     description: "用户管理接口文档"
6     email: zhanghuyi@itcast.cn
7     concat: 虎哥
8     url: https://www.itcast.cn
9     version: v1.0.0
10    group:
11      default:
12        group-name: default
13        api-rule: package
14        api-rule-resources:
15          - com.itheima.mp.controller
```

以后简单的增删改查可以直接在service层实现，就是上面的1-4接口，可以简单实现

```
1 package com.itheima.mp.domain.controller;
2
3 import cn.hutool.core.bean.BeanUtil;
4 import com.itheima.mp.domain.dto.UserFormDTO;
5 import com.itheima.mp.domain.po.User;
6 import com.itheima.mp.domain.vo.UserVO;
7 import com.itheima.mp.service.IUserService;
8 import io.swagger.annotations.Api;
9 import io.swagger.annotations.ApiOperation;
10 import io.swagger.annotations.ApiParam;
11 import lombok.AllArgsConstructor;
12 import lombok.NoArgsConstructor;
13 import org.apache.ibatis.annotations.Delete;
14 import org.springframework.web.bind.annotation.*;
15
16 import java.util.List;
17
18 @Api(tags = "用户管理接口")
19 @RequestMapping("/users")
20 @RestController
21 @NoArgsConstructor
22 public class UserController {
23
24     private final IUserService userService;
25
26
27     @ApiOperation("新增用户接口")
28     @PostMapping
29     public void saveUser(@RequestBody UserFormDTO userDTO){
30         //1.把dto拷贝到po
31         User user = BeanUtil.copyProperties(userDTO, User.class);
32         //2.新增
33         userService.save(user);
34     }
35
36     @ApiOperation("删除用户接口")
37     @DeleteMapping("{id}")
38     public void deleteUserById(@ApiParam("用户id") @PathVariable("id") Long id){
39         userService.removeById(id);
40     }
41
42     @ApiOperation("根据id查询用户接口")
43     @GetMapping("/{id}")
44 }
```

```

45     public UserVO queryUserById(@ApiParam("用户id") @PathVariable("id") Lo
46     ng id{
47         //查询用户PO
48         User user = userService.getById(id);
49         //把PO拷贝到VO
50         return BeanUtil.copyProperties(user, UserVO.class);
51     }
52
53     @ApiOperation("根据id批量查询用户id")
54     @GetMapping
55     public List<UserVO> queryUserById(@ApiParam("用户id集合") @RequestParam
("ids") List<Long> ids){
56         //查询用户PO
57         List<User> users = userService.listByIds(ids);
58         //把PO拷贝到VO
59         return BeanUtil.copyToList(users, UserVO.class);
60     }

```

但5则需要用到mapper层和service层

▼ controller层调用service接口

Java

```

1     @ApiOperation("扣减用户余额接口")
2     @PutMapping("/{id}/deduction/{money}")
3     public void deductMoneyById(
4             @ApiParam("用户id") @PathVariable("id") Long id,
5             @ApiParam("扣减的金额") @PathVariable("money") Integer money){
6             userService.deductBalance(id, money);
7         }

```

▼ 在service接口层实现接口定义

Java

```

1     public interface IUserService extends IService<User> {
2
3
4         void deductBalance(Long id, Integer money);
5     }

```

▼ 在service接口实现类实现接口，并调用mapper层

Java

```
1  @Service
2  public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements IUserService {
3      @Override
4      public void deductBalance(Long id, Integer money) {
5          //1.查询用户
6          User user= getById(id);
7          //2.校验用户状态
8          if(user==null || user.getStatus()==2){
9              throw new RuntimeException("用户状态异常");
10         }
11         //3.校验余额是否充足
12         if(user.getBalance() < money){
13             throw new RuntimeException("用户余额不足");
14         }
15         //4.扣减余额    update tb_user set balance = balance - ?
16         baseMapper.deductBalance(id,money);
17     }
18 }
```

▼ 在mapper层实现sql语句

Java

```
1  public interface UserMapper extends BaseMapper<User> {
2      List<User> queryUserByIds(@Param("id") List<Long> ids);
3
4      void updateBalanceByIds(@Param(Constants.WRAPPER) QueryWrapper<User> wrapper,@Param("amount") int amount);
5
6      @Update("update user set balance = balance - #{money} where id = #{id}")
7      void deductBalance(@Param("id") Long id,@Param("money") Integer money);
8  }
```

然后启动springboot项目，打开swagger，测试文档<http://localhost:8080/doc.html#/home>

测试开发功能

IService的Lambda查询

Lambda适用于帮助我们解决复杂的查询SQL语句的

需求：实现一个根据复杂条件查询用户的接口，查询条件如下：

- name：用户名关键字，可以为空
- status：用户状态，可以为空
- minBalance：最小余额，可以为空
- maxBalance：最大余额，可以为空

```
<select id="queryUsers" resultType="com.itheima.mp.domain.po.User">
    SELECT *
    FROM tb_user
    <where>
        <if test="name != null">
            AND username LIKE #{name}
        </if>
        <if test="status != null">
            AND `status` = #{status}
        </if>
        <if test="minBalance != null and maxBalance != null">
            AND balance BETWEEN #{minBalance} AND #{maxBalance}
        </if>
    </where>
</select>
```



```

1  @Override
2  @Transactional
3  public void deductBalance(Long id, Integer money) {
4      //1.查询用户
5      User user= getById(id);
6      //2.校验用户状态
7      if(user==null || user.getStatus()==2){
8          throw new RuntimeException("用户状态异常");
9      }
10     //3.校验余额是否充足
11     if(user.getBalance() < money){
12         throw new RuntimeException("用户余额不足");
13     }
14     //4.扣减余额    update tb_user set balance = balance - ?
15     int remainBalance = user.getBalance() - money;
16     lambdaUpdate()
17         .set(User::getBalance,remainBalance)
18         .set(remainBalance ==0,User::getStatus,2)
19         .eq(User::getId,id)
20         .update();
21 }
```

```

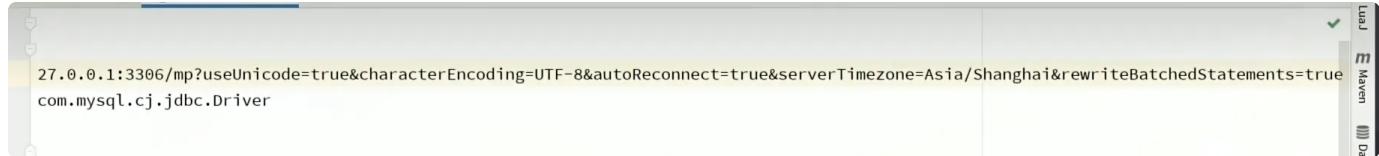
1  @Override
2  public List<User> queryUsers(String name, Integer status, Integer minBa
lance, Integer maxBalance) {
3      return lambdaQuery()
4          .like(name != null , User::getUsername , name)
5          .eq(status != null , User::getStatus , status)
6          .gt(minBalance != null , User::getBalance , minBalance)
7          .lt(maxBalance != null , User::getBalance , maxBalance)
8          .list() ;
9 }
```

IService批量新增（批处理）

需求：批量插入10万条用户数据，并作出对比：

- 普通for循环插入
- IService的批量插入
- 开启rewriteBatchedStatements=true参数

在配置文件中插入



27.0.0.1:3306/mp?useUnicode=true&characterEncoding=UTF-8&autoReconnect=true&serverTimezone=Asia/Shanghai&rewriteBatchedStatements=true
com.mysql.cj.jdbc.Driver

代码不改，插入配置就能大大减少时长

```
@Test
void testSaveBatch() {
    // 我们每次批量插入1000条件，插入100次即10万条数据

    // 1. 准备一个容量为1000的集合
    List<User> list = new ArrayList<>( initialCapacity: 1000);
    long b = System.currentTimeMillis();
    for (int i = 1; i <= 100000; i++) {
        // 2. 添加一个user
        list.add(buildUser(i));
        // 3. 每1000条批量插入一次
        if (i % 1000 == 0) {
            userService.saveBatch(list);
            // 4. 清空集合，准备下一批数据
            list.clear();
        }
    }
    long e = System.currentTimeMillis();
    System.out.println("耗时：" + (e - b));
}
```

批处理方案：

- 普通for循环逐条插入速度极差，不推荐
- MP的批量新增，基于预编译的批处理，性能不错
- 配置jdbc参数，开rewriteBatchedStatements，性能最好

代码生成（一次生成某表对应的四层框架）

一般流程：

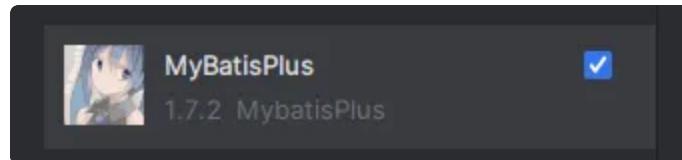
```
@TableName("user")
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    private String name;
    private Integer age;
    private Boolean isMarried;
    private Integer order;
}
```

```
public interface UserMapper extends BaseMapper<User> { }
```

```
public interface IUserService extends IService<User> { }
```

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements IUserService{ }
```

插件



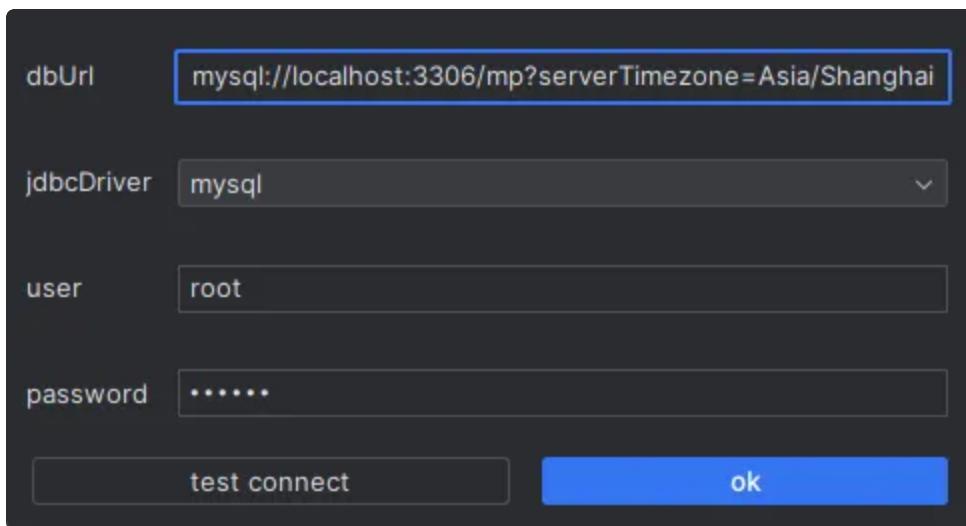


table name	create time	engine	coding	remark
address	2024-09-04 00:58:53	InnoDB	utf8mb3_general_ci	
user	2024-09-04 00:58:53	InnoDB	utf8mb3_general_ci	用户表

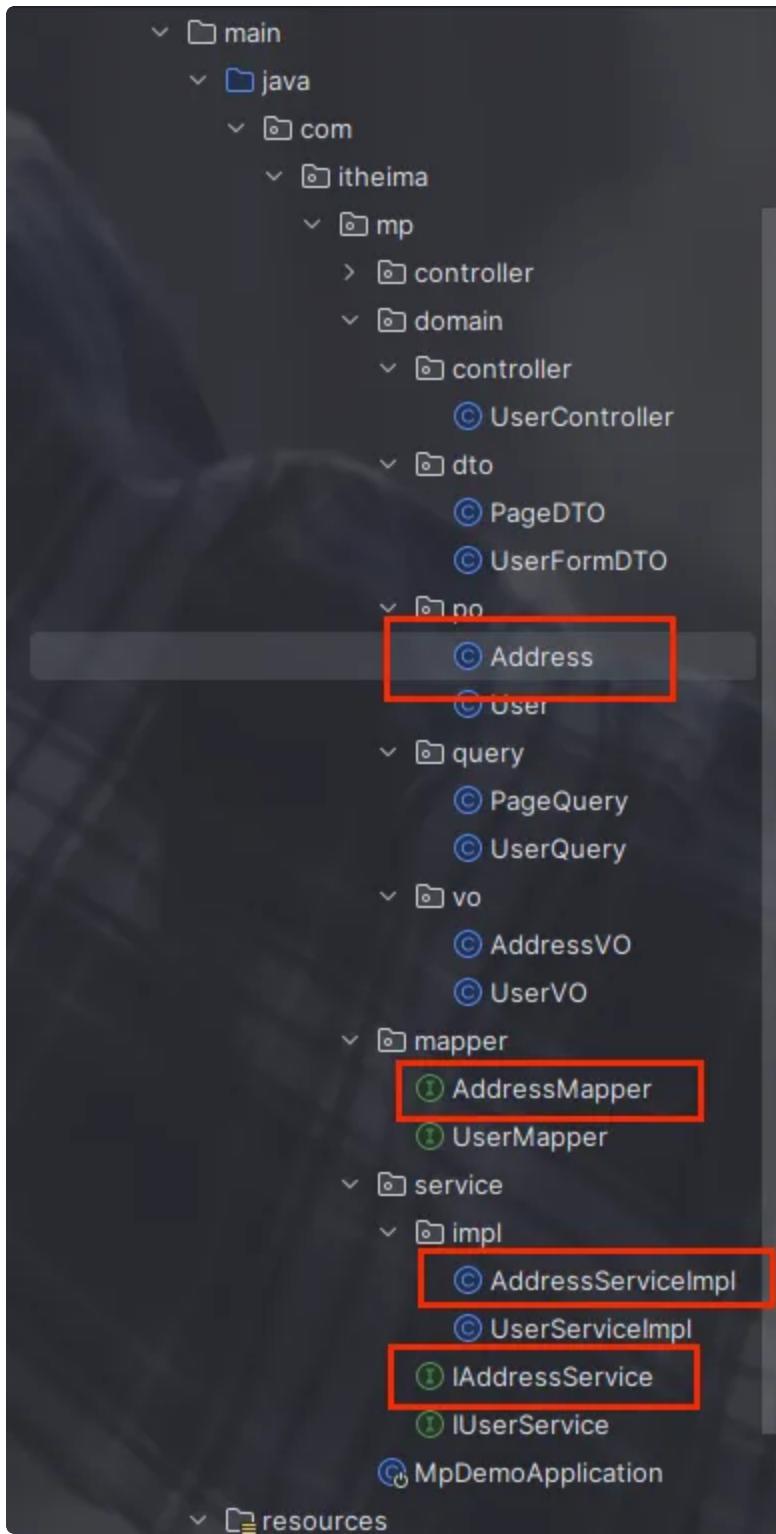
module: mp-demo package: com.itheima.mp

author: 虎哥 over file AUTO(ID自增)

Entity: domain.po Mapper: mapper Controller: controller

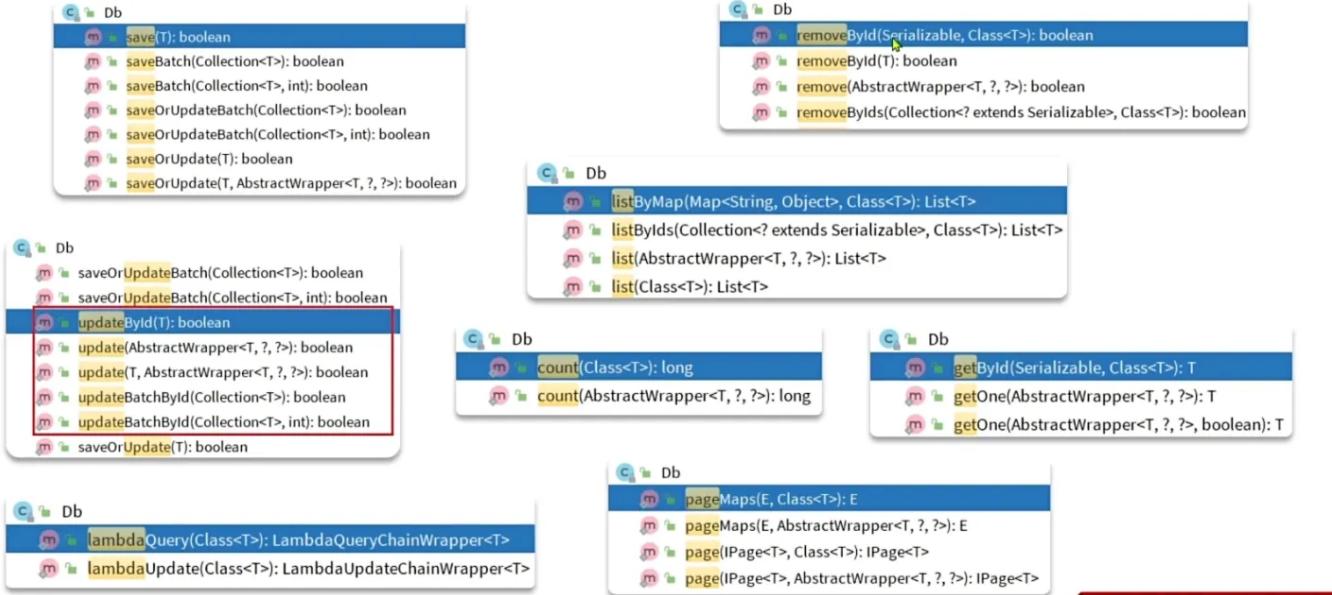
Service: service ServicImpl: service.impl TablePrefix:

lombok restController swagger ResultMap is fill is enable cache is base column



静态工具

DB静态工具，使用情况：service相互调用的情况下使用DB静态工具，既方便也避免了循环依赖



需求：

- ① 改造根据id查询用户的接口，查询用户的同时，查询出用户对应的所有地址
- ② 改造根据id批量查询用户的接口，查询用户的同时，查询出用户对应的所有地址
- ③ 实现根据用户id查询收货地址功能，需要验证用户状态，冻结用户抛出异常（练习）

1.

vo

```

1  @ApiModelProperty("用户收获地址")
2  private List<AddressV0> addresses;

```

controller

```

1  @ApiOperation("根据id查询用户接口")
2  @GetMapping("{id}")
3  public UserV0 queryUserById(@ApiParam("用户id") @PathVariable("id") Long id){
4      return userService.queryUserAndAddressById(id);
5  }

```

接口

```

1  UserV0 queryUserAndAddressById(Long id);

```

实现类

Java

```
1     @Override
2     public UserVO queryUserAndAddressById(Long id) {
3         //1.查询用户
4         User user = getById(id);
5         if(user == null || user.getStatus() == 2){
6             throw new RuntimeException("用户状态异常");
7         }
8         //2.查询地址
9         List<Address> addresses = Db.lambdaQuery(Address.class)
10            .eq(Address::getUserId, id).list();
11        //3.封装VO
12        //转user的po为vo
13        UserVO userVO = BeanUtil.copyProperties(user, UserVO.class);
14        //转地址vo
15        if(CollectionUtil.isNotEmpty(addresses)){
16            userVO.setAddresses(BeanUtil.copyToList(addresses, AddressVO.c
17                lass));
18        }
19        return userVO;
}
```

2、

```

1      @Override
2      public List<UserV0> queryUserAndAddressByIds(List<Long> ids) {
3          //1.查询用户
4          List<User> users = listByIds(ids);
5          if(CollUtil.isEmpty(users)){
6              return Collections.emptyList();
7          }
8          //2.查询地址
9          //获取用户id集合
10         List<Long> userIds = users.stream().map(User::getId).collect(Collectors.toList());
11         //根据用户id查询地址
12         List<Address> addresses = Db.lambdaQuery(Address.class).in(Address
13 ::getUserId, userIds).list();
14         //转换地址vo
15         List<AddressV0> addressV0List = BeanUtil.copyToList(addresses, AddressV0.class);
16         //用户地址集合分组处理，相同用户的放入同一个集合组中
17         Map<Long,List<AddressV0>> addressMap = new HashMap<>(0);
18         if(CollUtil.isNotEmpty(addressV0List)){
19             addressMap = addressV0List.stream().collect(Collectors.groupingBy(
20             AddressV0::getUserId));
21         }
22
23         //转换V0返回
24         List<UserV0> list = new ArrayList<>(users.size());
25         for(User user : users){
26             //转换user的po为vo
27             UserV0 vo = BeanUtil.copyProperties(user, UserV0.class);
28             list.add(vo);
29             //转换地址vo
30             vo.setAddresses(addressMap.get(user.getId()));
31         }
32     }

```

逻辑删除

逻辑删除就是基于代码逻辑模拟删除效果，但并不会真正删除数据。思路如下：

- 在表中添加一个字段标记数据是否被删除
- 当删除数据时把标记置为1
- 查询时只查询标记为0的数据

例如逻辑删除字段为deleted：

- 删除操作：

```
UPDATE user SET deleted = 1 WHERE id = 1 AND deleted = 0
```

- 查询操作：

```
SELECT * FROM user WHERE deleted = 0
```

MybatisPlus提供了逻辑删除功能，无需改变方法调用的方式，而是在底层帮我们自动修改CRUD的语句。我们要做的就是在application.yaml文件中配置逻辑删除的字段名称和值即可：

```
mybatis-plus:  
  global-config:  
    db-config:  
      logic-delete-field: flag # 全局逻辑删除的实体字段名，字段类型可以是boolean、integer  
      logic-delete-value: 1 # 逻辑已删除值(默认为 1)  
      logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

注意

逻辑删除本身也有自己的问题，比如：

- 会导致数据库表垃圾数据越来越多，影响查询效率
- SQL中全都需要对逻辑删除字段做判断，影响查询效率

因此，我不太推荐采用逻辑删除功能，如果数据不能删除，可以采用把数据迁移到其它表的办法。

```
1     @Test
2     void testLongDelete(){
3         //1.删除
4         addressService.removeById(59L);
5         //2.查询
6         Address address = addressService.getById(59L);
7         System.out.println("address = " + address);
8     }
```

枚举处理器

User类中有一个用户状态字段：

The screenshot shows a Java code editor with the file 'User.java' open. The code contains two private fields: 'info' and 'status'. The 'status' field is highlighted with a red rounded rectangle. To the right of the code, there is explanatory text in Chinese: '详细信息' (Detailed information) above the 'info' field, and '使用状态 (1正常 2冻结)' (Usage status (1Normal 2Frozen)) above the 'status' field.

```
38     private String info;
39
40     private Integer status;
```

```

@Getter
public enum UserStatus {
    NORMAL(1, "正常"),
    FREEZE(2, "冻结")
;

    private final int value;
    private final String desc;

    UserStatus(int value, String desc) {
        this.value = value;
        this.desc = desc;
    }
}

```

枚举类型要

和数据库中的类型相互转化

#	名称	数据类型	注释
1	id	BIGINT	用户id
2	username	VARCHAR	用户名
3	password	VARCHAR	密码
4	phone	VARCHAR	注册手机号
5	info	JSON	详细信息
6	status	INT	使用状态 (1正常 2冻结)
7	balance	INT	账户余额

mbp的处理器:

* I TypeHandler (org.apache.ibatis.type)

- ✓ C BaseTypeHandler (org.apache.ibatis.type)
 - C BlobTypeHandler (org.apache.ibatis.type)
 - C DateTypeHandler (org.apache.ibatis.type)
 - C IntegerTypeHandler (org.apache.ibatis.type)
 - C SqlTimeTypeHandler (org.apache.ibatis.type)
 - C ArrayTypeHandler (org.apache.ibatis.type)
 - C StringTypeHandler (org.apache.ibatis.type)
 - C EnumOrdinalTypeHandler (org.apache.ibatis.type)
 - C BigDecimalTypeHandler (org.apache.ibatis.type)
 - C BooleanTypeHandler (org.apache.ibatis.type)
 - C ObjectTypeHandler (org.apache.ibatis.type)
 - C DoubleTypeHandler (org.apache.ibatis.type)
 - C ShortTypeHandler (org.apache.ibatis.type)
 - C LongTypeHandler (org.apache.ibatis.type)
 - C LocalDateTypeHandler (org.apache.ibatis.type)
 - C UnknownTypeHandler (org.apache.ibatis.type)
 - C BigIntegerTypeHandler (org.apache.ibatis.type)
 - C ByteArrayTypeHandler (org.apache.ibatis.type)
 - C MybatisEnumTypeHandler (com.baomidou.mybatisplus.core.handlers)
 - C ClobTypeHandler (org.apache.ibatis.type)
 - C SqlxmlTypeHandler (org.apache.ibatis.type)
- > C AbstractJsonTypeHandler (com.baomidou.mybatisplus.extension.handlers)

为实现转化，加上注解

```
@Getter  
public enum UserStatus {  
    NORMAL(1, "正常"),  
    FREEZE(2, "冻结")  
;  
    @EnumValue  
    private final int value;  
    private final String desc;  
  
    UserStatus(int value, String desc) {  
        this.value = value;  
        this.desc = desc;  
    }  
}
```

① 给枚举中的与数据库对应value值添加@EnumValue注解

```
@EnumValue  
private final int value;  
@JsonValue  
private final String desc;
```

配置mbp，让注解生效

```
mybatis-plus:  
    configuration:  
        default-enum-type-handler: com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler
```

po、vo修改

```
💡 @ApiModelProperty("使用状态(1正常 2冻结)")  
private UserStatus status;
```

userservice修改

```
//1.查询用户  
User user= getById(id);  
//2.校验用户状态  
if(user==null ||user.getStatus()== UserStatus.FROZEN){  
    throw new RuntimeException("用户状态异常");  
}
```

JSON处理器

数据库中user表中有一个json类型的字段：

#	名称	数据类型	注释	长度/集合
1	id	BIGINT	用户id	19
2	username	VARCHAR	用户名	50
3	password	VARCHAR	密码	128
4	phone	VARCHAR	注册手机号	20
5	info	JSON	详细信息	
6	status	INT	使用状态(1正常 2冻结)	10
7	balance	INT	账户余额	10
8	create_time	DATETIME	创建时间	
9	update_time	DATETIME	更新时间	



```
@Data  
@TableName("user")  
public class User {  
    private Long id;  
  
    private String username;  
  
    private String info;
```

```
@Data  
@TableName("user")  
public class User {  
    private Long id;  
  
    private String username;  
  
    private UserInfo info;
```

```
@Data  
public class UserInfo {  
    private Integer age;  
    private String intro;  
    private String gender;
```

Java

```
1 package com.itheima.mp.domain.po;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @NoArgsConstructor
9 @AllArgsConstructor(staticName = "of")
10 public class UserInfo {
11     private Integer age;
12     private String intro;
13     private String gender;
14 }
```

Java

```
1 @ApiModelProperty("详细信息")
2 private UserInfo info;
3
4
5
6
7 /**
8 * 详细信息
9 */
10 @TableField(typeHandler = JacksonTypeHandler.class)
11 private UserInfo info;
```

```

1  @Test
2  void testInsert() {
3      User user = new User();
4      user.setId(5L);
5      user.setUsername("Lucy");
6      user.setPassword("123");
7      user.setPhone("18688990011");
8      user.setBalance(200);
9      //      user.setInfo("{\"age\": 24, \"intro\": \"英文老师\", \"gender\": \"female\"}");
10     user.setInfo(UserInfo.of(24,"英文老师","female"));
11     user.setCreateTime(LocalDateTime.now());
12     user.setUpdateTime(LocalDateTime.now());
13     userMapper.insert(user);
14 }

```

插件功能-分页插件基本用法

MyBatisPlus提供的内置拦截器有下面这些：

序号	拦截器	描述
1	TenantLineInnerInterceptor	多租户插件
2	DynamicTableNameInnerInterceptor	动态表名插件
3	PaginationInnerInterceptor	分页插件
4	OptimisticLockerInnerInterceptor	乐观锁插件
5	IllegalSQLInnerInterceptor	SQL性能规范插件，检测并拦截垃圾SQL
6	BlockAttackInnerInterceptor	防止全表更新和删除的插件

分页插件

首先，要在配置类中注册MyBatisPlus的核心插件，同时添加分页插件：

```
@Configuration
public class MybatisConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        // 1. 初始化核心插件
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        // 2. 添加分页插件
        PaginationInnerInterceptor pageInterceptor = new PaginationInnerInterceptor(DbType.MYSQL);
        pageInterceptor.setMaxLimit(1000L); // 设置分页上限
        interceptor.addInnerInterceptor(pageInterceptor);
        return interceptor;
    }
}
```

接着，就可以使用分页的API了：



```
@Test
void testPageQuery() {
    // 1. 查询
    int pageNo = 1, pageSize = 5;
    // 1.1. 分页参数
    Page<User> page = Page.of(pageNo, pageSize);
    // 1.2. 排序参数，通过OrderItem来指定
    page.addOrder(new OrderItem("balance", false));
    // 1.3. 分页查询
    Page<User> p = userService.page(page);
    // 2. 总条数
    System.out.println("total = " + p.getTotal());
    // 3. 总页数
    System.out.println("pages = " + p.getPages());
    // 4. 分页数据
    List<User> records = p.getRecords();
    records.forEach(System.out::println);
}
```

配插件

Java

```
1 package com.itheima.mp.config;
2
3 import com.baomidou.mybatisplus.annotation.DbType;
4 import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
5 import com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInt
erceptor;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 @Configuration
10 public class MyBatisConfig {
11     @Bean
12     public MybatisPlusInterceptor mybatisPlusInterceptor(){
13         MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
14         //1. 创建分页插件
15         PaginationInnerInterceptor paginationInnerInterceptor = new Pagina
tionInnerInterceptor(DbType.MYSQL);
16         paginationInnerInterceptor.setMaxLimit(1000L);
17         //2. 添加分页插件
18         interceptor.addInnerInterceptor(paginationInnerInterceptor);
19         return interceptor;
20     }
21 }
22 }
```

▼ 使用插件

Java |

```
1     @Test
2     void testPageQuery(){
3         int pageNo = 1, pageSize = 2;
4         // 1.准备分页条件
5         Page<User> page = new Page<>(pageNo, pageSize);
6         page.addOrder(OrderItem.asc("balance"));
7         // 2.分页查询
8         Page<User> p = userService.page(page);
9         // 3.解析
10        long total = p.getTotal(); // 总记录数
11        System.out.println("total = " + total);
12        long pages = p.getPages(); // 总页数
13        System.out.println("pages = " + pages);
14        List<User> users = p.getRecords(); // 当前页的记录
15        users.forEach(System.out::println);
16    }
```

通用分页实体

简单分页查询案例

需求：遵循下面的接口规范，编写一个UserController接口，实现User的分页查询

参数	说明
请求方式	GET
请求路径	/users/page
请求参数	{ "pageNo": 1, "pageSize": 5, "sortBy": "balance", "isAsc": false, "name": "jack", "status": 1 }
返回值	
特殊说明	<ul style="list-style-type: none">如果排序字段为空，默认按照更新时间排序排序字段不为空，则按照排序字段排序

```
{  
  "total": 1005,  
  "pages": 201,  
  "list": [  
    {  
      "id": 1,  
      "username": "Jack",  
      "info" : {  
        "age": 21,  
        "gender": "male",  
        "intro": "佛系青年"  
      },  
      "status": "正常",  
      "balance": 2000  
    },  
    {  
      "id": 2,  
      "username": "Rose"  
    }  
  ]}
```

```
        "username": "rose",
        "info": {
            "age": 20,
            "gender": "female",
            "intro": "文艺青年"
        },
        "status": "冻结",
        "balance": 1000
    }
]
```

创建pageDTO

```
1 package com.itheima.mp.domain.dto;
2
3 import cn.hutool.core.bean.BeanUtil;
4 import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
5 import lombok.AllArgsConstructor;
6 import lombok.Data;
7 import lombok.NoArgsConstructor;
8
9 import java.util.Collections;
10 import java.util.List;
11 import java.util.function.Function;
12 import java.util.stream.Collectors;
13
14 @Data
15 @NoArgsConstructor
16 @AllArgsConstructor
17 public class PageDTO<V> {
18     private Long total;
19     private Long pages;
20     private List<V> list;
21
22     /**
23      * 返回空分页结果
24      * @param p MybatisPlus的分页结果
25      * @param <V> 目标VO类型
26      * @param <P> 原始PO类型
27      * @return VO的分页对象
28      */
29     public static <V, P> PageDTO<V> empty(Page<P> p){
30         return new PageDTO<>(p.getTotal(), p.getPages(), Collections.emptyList());
31     }
32
33     /**
34      * 将MybatisPlus分页结果转为 VO分页结果
35      * @param p MybatisPlus的分页结果
36      * @param voClass 目标VO类型的字节码
37      * @param <V> 目标VO类型
38      * @param <P> 原始PO类型
39      * @return VO的分页对象
40      */
41     public static <V, P> PageDTO<V> of(Page<P> p, Class<V> voClass) {
42         // 1.非空校验
43         List<P> records = p.getRecords();
44         if (records == null || records.size() <= 0) {
```

```
45         // 无数据, 返回空结果
46         return empty(p);
47     }
48     // 2.数据转换
49     List<V> vos = BeanUtil.copyToList(records, voClass);
50     // 3.封装返回
51     return new PageDTO<>(p.getTotal(), p.getPages(), vos);
52 }
53
54 /**
55 * 将MybatisPlus分页结果转为 VO分页结果, 允许用户自定义PO到VO的转换方式
56 * @param p MybatisPlus的分页结果
57 * @param convertor PO到VO的转换函数
58 * @param <V> 目标VO类型
59 * @param <P> 原始PO类型
60 * @return VO的分页对象
61 */
62 public static <V, P> PageDTO<V> of(Page<P> p, Function<P, V> convertor
) {
63     // 1.非空校验
64     List<P> records = p.getRecords();
65     if (records == null || records.size() <= 0) {
66         // 无数据, 返回空结果
67         return empty(p);
68     }
69     // 2.数据转换
70     List<V> vos = records.stream().map(convertor).collect(Collectors.t
oList());
71     // 3.封装返回
72     return new PageDTO<>(p.getTotal(), p.getPages(), vos);
73 }
74 }
75 }
```

创建pagequery

```

1  @Data
2  @ApiModel(description = "分页查询实体")
3  public class PageQuery {
4      @ApiModelProperty("页码")
5      private Integer pageNo;
6      @ApiModelProperty("页码")
7      private Integer pageSize;
8      @ApiModelProperty("排序字段")
9      private String sortBy;
10     @ApiModelProperty("是否升序")
11     private Boolean isAsc;
12 }
```

userquery继承pagequery

在controller层

```

1  @ApiOperation("根据条件分页查询用户接口")
2  @PutMapping("/page")
3  public PageDTO<UserVO> queryUsersPage(UserQuery query){
4      return userService.queryUsersPage(query);
5 }
```

service实现类

```

1  @Override
2  public PageDTO<UserVO> queryUsersPage(UserQuery query) {
3      String name = query.getName();
4      Integer status = query.getStatus();
5      // 1.构建分页条件
6      Page<User> page = Page.of(query.getPageNo(), query.getPageSize());
7      if(StrUtil.isNotBlank(query.getSortBy())){
8          // 不为空
9          page.addOrder(new OrderItem(query.getSortBy(), query.getIsAsc()
10         )));
11     } else {
12         // 为空, 按默认更新时间排序
13         page.addOrder(new OrderItem("update_time", false));
14     }
15
16     // 2.分页查询
17     Page<User> p = lambdaQuery()
18         .like(name != null, User::getUsername, name)
19         .eq(status != null, User::getStatus, status)
20         .page(page);
21
22     // 3.封装VO结果
23     PageDTO<UserVO> dto = new PageDTO<>(); // 修改为 PageDTO<UserVO>
24     // 总条数
25     dto.setTotal(p.getTotal());
26     // 总页数
27     dto.setPages(p.getPages());
28     // 当前页数据
29     List<User> records = p.getRecords();
30     if(CollectionUtil.isEmpty(records)){
31         dto.setList(Collections.emptyList());
32         return dto;
33     }
34     // 拷贝 user 到 VO
35     dto.setList(BeanUtil.copyToList(records, UserVO.class));
36     // 4.返回
37     return dto;
38 }
```

通用分页实体与MP转换

```
1 package com.itheima.mp.domain.query;
2
3 import com.baomidou.mybatisplus.core.metadata.OrderItem;
4 import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
5 import io.swagger.annotations.ApiModel;
6 import io.swagger.annotations.ApiModelProperty;
7 import lombok.Data;
8
9 @Data
10 @ApiModel(description = "分页查询实体")
11 public class PageQuery {
12     @ApiModelProperty("页码")
13     private Integer pageNo;
14     @ApiModelProperty("页码")
15     private Integer pageSize;
16     @ApiModelProperty("排序字段")
17     private String sortBy;
18     @ApiModelProperty("是否升序")
19     private Boolean isAsc;
20     public <T> Page<T> toMpPage(OrderItem ... orders){
21         // 1.分页条件
22         Page<T> p = Page.of(pageNo, pageSize);
23         // 2.排序条件
24         // 2.1.先看前端有没有传排序字段
25         if (sortBy != null) {
26             p.addOrder(new OrderItem(sortBy, isAsc));
27             return p;
28         }
29         // 2.2.再看有没有手动指定排序字段
30         if(orders != null){
31             p.addOrder(orders);
32         }
33         return p;
34     }
35
36     public <T> Page<T> toMpPage(String defaultSortBy, boolean isAsc){
37         return this.toMpPage(new OrderItem(defaultSortBy, isAsc));
38     }
39
40     public <T> Page<T> toMpPageDefaultSortByCreateTimeDesc() {
41         return toMpPage("create_time", false);
42     }
43
44     public <T> Page<T> toMpPageDefaultSortByUpdateTimeDesc() {
45         return toMpPage("update_time", false);
```

46 }
47 }
48

```
1 package com.itheima.mp.domain.dto;
2
3 import cn.hutool.core.bean.BeanUtil;
4 import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
5 import lombok.AllArgsConstructor;
6 import lombok.Data;
7 import lombok.NoArgsConstructor;
8
9 import java.util.Collections;
10 import java.util.List;
11 import java.util.function.Function;
12 import java.util.stream.Collectors;
13
14 @Data
15 @NoArgsConstructor
16 @AllArgsConstructor
17 public class PageDTO<V> {
18     private Long total;
19     private Long pages;
20     private List<V> list;
21
22     /**
23      * 返回空分页结果
24      * @param p MybatisPlus的分页结果
25      * @param <V> 目标VO类型
26      * @param <P> 原始PO类型
27      * @return VO的分页对象
28      */
29     public static <V, P> PageDTO<V> empty(Page<P> p){
30         return new PageDTO<V>(p.getTotal(), p.getPages(), Collections.emptyList());
31     }
32
33     /**
34      * 将MybatisPlus分页结果转为 VO分页结果
35      * @param p MybatisPlus的分页结果
36      * @param voClass 目标VO类型的字节码
37      * @param <V> 目标VO类型
38      * @param <P> 原始PO类型
39      * @return VO的分页对象
40      */
41     public static <V, P> PageDTO<V> of(Page<P> p, Class<V> voClass) {
42         // 1.非空校验
43         List<P> records = p.getRecords();
44         if (records == null || records.size() <= 0) {
```

```
45         // 无数据, 返回空结果
46         return empty(p);
47     }
48     // 2.数据转换
49     List<V> vos = BeanUtil.copyToList(records, voClass);
50     // 3.封装返回
51     return new PageDTO<>(p.getTotal(), p.getPages(), vos);
52 }
53
54 /**
55 * 将MybatisPlus分页结果转为 VO分页结果, 允许用户自定义PO到VO的转换方式
56 * @param p MybatisPlus的分页结果
57 * @param convertor PO到VO的转换函数
58 * @param <V> 目标VO类型
59 * @param <P> 原始PO类型
60 * @return VO的分页对象
61 */
62 public static <V, P> PageDTO<V> of(Page<P> p, Function<P, V> convertor
) {
63     // 1.非空校验
64     List<P> records = p.getRecords();
65     if (records == null || records.size() <= 0) {
66         // 无数据, 返回空结果
67         return empty(p);
68     }
69     // 2.数据转换
70     List<V> vos = records.stream().map(convertor).collect(Collectors.t
oList());
71     // 3.封装返回
72     return new PageDTO<>(p.getTotal(), p.getPages(), vos);
73 }
74 }
75 }
```